Advanced Bit Operations

Boolean Operations

The 80X86 CPU implements the logical **BOOLEAN** operations **AND**, **OR**, **XOR** which are applied between corresponding bits between two words and **NOT** which is applied to the individual bits of a single word:

and/or/xor	***reg/mem, reg/mem/constant***
not	***reg/mem***

with **not more than one** operand from memory

e.g. Assume:
AX = 0001 0010 0011 0100 B
BX = 1111 1110 1101 0011 B

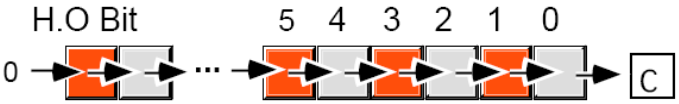
AND ax, bx will yield
AX = 0001 0010 00**0**1 0**0**00 **B** **(AX changed bits in bold)**
BX = 1111 1110 1101 0011 B **(BX unchanged)**
OR ax, bx will yield
AX = 1111 **11**10 **11**11 01**11 B** **(AX changed bits in bold)**
BX = 1111 1110 1101 0011 B **(BX unchanged)**
XOR ax, bx will yield
AX = 1110 1100 **1110** 01**11 B** **(AX changed bits in bold)**
BX = 1111 1110 1101 0011 B **(BX unchanged)**
NOT ax will yield
AX = **1110 1101 1100 1011** B **(AX changed bits in bold - All of them)**

SHIFT OPERATIONS

Logical shifts

shl (shift-left)

shr (shift-right)



Syntax:

shl/shr	**mem/reg, cl/constant**

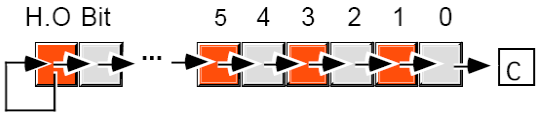
Thus if ax = 1101 0010 1101 0001B and cl contains 3

shr	ax, 1	ax = **0**110 1001 0110 1000	CF = 1
shl	ax, cl	ax = 1001 0110 1000 1**000**	CF = 0

Arithmetic shifts

sal (shift-arithmetic left) same as **shl**

sar (shift-arithmetic right)

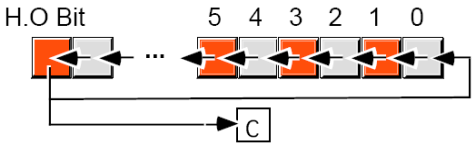


Syntax: **sal/sar** **mem/reg, cl/constant**

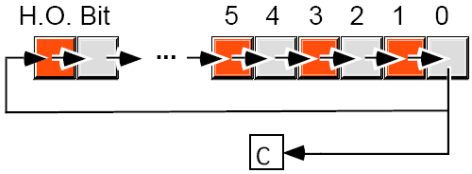
SAR is used to preserve sign on signed operations.

Rotates

rol (rotate-left)



ror (rotate-right)



Syntax: **rol/ror** **mem/reg, cl/constant**

Floating Point Unit

FPU Data

We will now address real (**floating point**) numbers. There is a special unit called the **Floating Point Unit**, or **FPU**. On some CPUs of the Intel family, the FPU is physically separate known as the **numeric coprocessor**. The floating point numbers can be in one of five separate forms:

Floating Point Formats

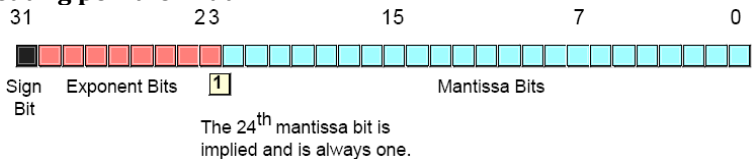
type	C equivalent	size (in bits)
short floating point	float	32
long floating point	double	64
integer (DW)	short int	16
long integer (DD)	long int	32
Binary Coded Decimal (BCD)	NONE	4
*extended floating point	NONE	80

* Internal format only

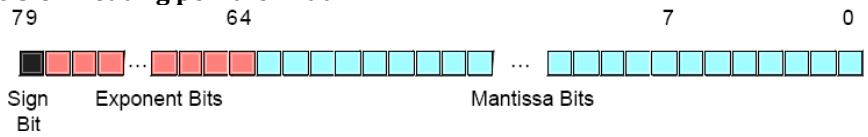
Floating Point Numbers

If the integer number 100 is 1 X 10^2, then the number 123 is 1.23 X 10^2. This is referred to as scientific notation. It lets us represent very large or very small numbers without worrying about a lot of digits in the number. To make it more complex, we can represent the number -0.00123 as -1.23X 10^{-3}, or -1.23E-3 expressed in binary, of course.

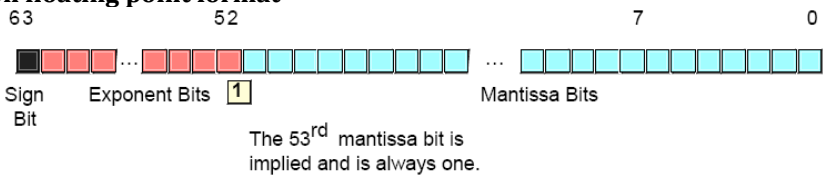
32-bit Single precision floating point format



80-bit Extended precision floating point format



64-bit Double precision floating point format



- The sign bit (1 if negative and zero if positive. (bit 63)
- The exponent bits (the powers of two!!!). (bits 52-62)
- The mantissa. (bits 0 - 51)

The exponent is biased. That is value is in the range of -1023 to 1024. The **bias** is 1023, when added to the exponent results in a positive number. (If the exponent is 3, then 1026 is stored as the exponent portion of the floating point number.)

The mantissa is the part of the number that holds the value we are talking about, except that it is **normalized**. In scientific notation, all numbers are multiplied or divided by ten so that there is only one digit in front of the decimal sign. The exponent is adjusted to compensate for that. 1.23 X 10^2 is the equivalent of 1.23 X 100. (10^2 is 100). 1.23 X 100 is 123 when you multiple it out.

In the binary system, we change the power of 10 to the power of 2. This means that if we have a non-zero digit in front of decimal point, it must be a 1. Therefore, we can assume that it is present!

This means the number is represented as: $\mathbf{((-1)^s \text{ X } 1.b_{51}b_{50}b_{49}...b_2b_1b_0 \text{ X } 2^{(exp-1023)})}$

b_{51} = 0.5 if set,
b_{50} = 0.25 if set,
b_{49} = 0.125 if set, etc.

Range of Floating Point Numbers

Binary digits and exponents do not correspond exactly to decimal digits and exponent, but the following are the approximate ranges of floating point numbers in decimal:

	decimal digits of precision	range of exponent
short	7	10^{-38} to 10^{38}
long	15	10^{-308} to 10^{308}
extended	19	10^{-4932} to 10^{4932}

There are two terms that we must understand here, precision and accuracy Precision is the number of digits represented starting with the first non-zero number. 123, -123, 123000, 0.123, and -0.000123 all have a precision of 3. Accuracy is the total number of digits in the value you are trying to represent correctly. 3.140000 could be a seven digit precision but with only 3 place accuracy.

Declaring Floating Point Variables

Short floating point variable are declared using the **DD** data type:

```
A   DD    -1.23      ; A negative short fp number
B   DD    54E23      ; A positive short fp number
C   DD    -5432      ; A negative long integer
D   DD    ?          ; A variable which can hold either a short fp or long integer
E   DQ    -1.23      ; A negative long fp number
```

Floating Point Instructions

All floating point operations, and **only** floating point operations, start with an initial *f*.

f*operation* zero or more operands

For BCD and integer operands, there is a second letter:

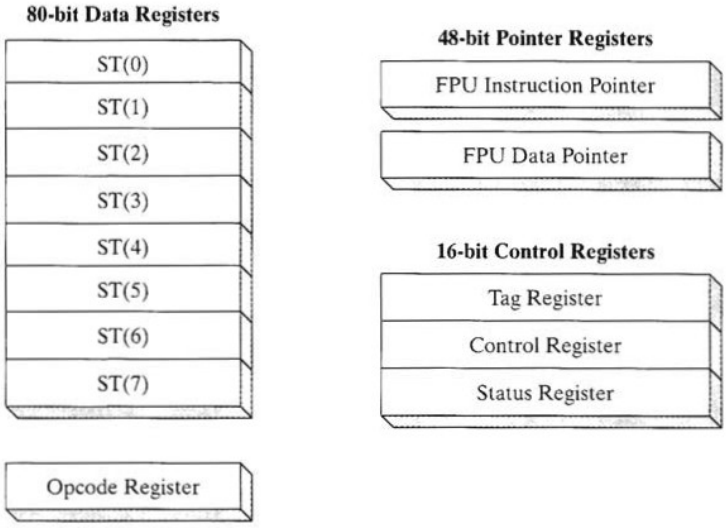
*f***b***operation* operand ;**BCD** operand
*f***i***operation* operand ;**Integer** operand

All other second letters are for floating point arguments.
If the last letter is a '**p**', the FPU stack is popped on completion of the operation.

*foperation***p** zero or more operands

The FPU stack

The most important registers in the FPU are a stack of eight extended floating point numbers and a 2-bit tag field for each of the eight. One of the registers is designated as the top of the stack (ST) and all other registers are relative to it: ST(1), ST(2). Data is pushed on the stack with a **load**. In every FPU computation, one of the operands is ST. If an attempt to put value on a non-empty entry will result in an error.



FPU Load (Push)

```
fld     mem32 or mem64      ; short or long fp load
fld     ST(n)               ; load from another stack entry
fild    mem16 or mem32      ; int or long int load
```

FPU Load (Push) constants

```
fld1        ; load 1
fldz        ; load zero
fldpi       ; load p
fldl2t      ; load log2(10)
fldl2e      ; load log2(e)
fldlg2      ; load log10(2)
fldln2      ; load loge(2)
```

FPU Store (and pop) ST

```
fst[p]    mem32 or mem64    ; store [and pop] ST
fst[p]    ST(n)             ; copy [then pop] ST to ST(n)
fist[p]   mem16 or mem32    ; store [and pop] ST as int
```

FPU Miscellaneous Stack Instructions

```
fcomp   ST      ; pop FPU stack
fld     ST      ; load duplicate of stack top
fxch    ST(n)   ; exchange contents of ST and ST(n)
fxch            ; exchange contents of ST and ST(1)
```

FPU Arithmetic

The FPU is a **stack machine**. Arithmetic is performed by pushing the two operands onto the stack, and then executing an opcode, which in effect pops the two top items, does the operation and then pushes the result back on the stack. You can use the letter **p** optionally appended to pop the result.

```
fadd    ST + ST(1)      fmul    ST X ST(1)
fsub    ST - ST(1)      fdiv    ST / ST(1)
fsubr   ST(1) - ST      fdivr   ST(1) / ST
```

FPU Input/Output

There are two routines in UTIL.LIB to do this for you:

Name	input	output
GetFP	none	floating point number from keyboard is in ST
PutFP	ST on the FPU stack	number is display and popped from ST

FPU Compare & Branching

The comparisons will be done in the FPU and the decision to branch will be done in the CPU. This must be done in two steps:

FPU Compare Instructions

```
fcom[p]    mem32 or mem64    ; compare ST to operand [and pop]
fcom[p]    ST(n)             ; compare ST to ST(n) [and pop]
fcom[p]                      ; compare ST to ST(1) [and pop]
fcompp                       ; compare ST to ST(1) and pop twice
ficom[p]   mem16 or mem32    ; compare ST to converted integer operand [and pop]
ftst                         ; compare ST to 0.0
```

Transferring the FPU Status Word to Flags

The FPU will compare and set bits in an FPU register (**status word**). These must then be transferred to the Flags Register in the CPU.

```
StatWd     DW      ?          ; defined in the .DATA Segment
           ...
           fld     B          ; Load B onto ST
           fld     C          ; Load C onto ST
           fcom               ; Compare B & C
           fstsw   StatWd     ; Store status word in StatWd
           mov     ax, StatWd ; Move status word in ax
           sahf               ;Transfer ah into flags register
           jbe     OrderOK    ;Is B <= C? if yes goto OrderOK
           fxch               ;otherwise swap  B & C
OrderOK:
           fstp    A
           fcomp   ST         ; pop smaller item
```